.

Eric Cuiffo, Nick Palumbo, Val A. Red, Aiser Sheikh
6 May 2013
198:416 Operating Systems, A03.
Simple Network File System

Introduction

Our group, consisting of Eric Cuiffo, Nick Palumbo, Val A. Red, and Aiser Sheikh, built a the client and server applications for a ***Simple Network File System (SNFS)*** . We accomplished this utilizing separate executables, `serverSNFS` and `clientSNFS`. The former, server-side executable is a multithreaded program that handles file management and requests; the latter, client-side executable communicates between user-end applications and the server over IP/TCP protocols. This utilizes FUSE to interpret and relay file system operations over network, then receiving the server's response to client-side applications.

I. Our Work on the SNFS Implementation :

For communication between the client and server-side applications, we decided a major point of our implementation would be to send data over raw sockets by converting all messages to the `char*` format. This appropriately fit the descriptor of *simple* network file system by being the most elegant method to transfer data and requests over raw sockets. As such, much of our code consists of simple character array and value manipulations corresponding to the data transferred over sockets.

`serverSNFS.c`

Our implementation of serverSNFS polls as a `while(1)` loop, polling for any requests sent over a designated port number to pass to the **request handler.** The request handler itself is a series of if-else statements that parse the the first argument to interpret what file system function is being requested.

Each argument required by the server to implement each function is delimited by commas, our implementation then tokenizes this to truncate the arguments.

Multithreading is handled such that each time a request is sent to the server it dynamically spawns a new thread calling our `request_handler` function, which then receives the data and switches on the name of the system call. As such, **the first argument sent over the socket is *always* the system call.**

`clientSNFS.c`

FUSE, **F**ilesystem in **U**ser**S**pac**E**, was a large part of our clientSNFS implementation, as it enables us to efficiently and securely implement a client application in the userspace that can remotely mount the filesystem on serverSNFS. Much of our code mediates between default FUSE operations and our serverSNFS implementation, enabling anyone in the user-end running clientSNFS to use file operations remotely.

As such, clientSNFS features analagous file operations to those on serverSNFS via FUSE. Consistent with our implementation for serverSNFS, it utilizes the same `char*` implementation for core functions, also extensively utilizing character-string manipulation functions.

II. Difficulties Faced

Most of the difficulties we faced were regarding memory management, as at several of points we would reach segmentation faults. As such, we were certain to build our program incrementally and avoid losing track of which part of our code was responsible for said segmentation faults. We were especially careful after implementing multithreading to check for any possible issues with our code. Particularly, `readdir` and `opendir` were tricky to implement with multithreading, as they would begin to cause segmentation faults while they worked fine when single-threaded. Nevertheless, after correcting issues with the order of our `char*` and `str` operations, we were able to debug the issues and fully implement those functions without error on execution.

One of the challenges we faced was attempting to use `memcpy` to change structs into `char*` and send them over to the client and vice versa. As the data was sent over, only empty structs resulted. We eventually decided on using strcpy and sending the data as strings because it was the easiest way although slightly less efficient. Nevertheless, the tradeoff of better management of data and bypassing this issue we encountered made it an elegant solution, again stressing the value of our Simple Network File System.

III. Use cases

To initialize the serverSNFS application, one can enter the following on a terminal in the `serverSNFS` directory:

```
./serverSNFS -mount FSDirectory/ -port 6590
```

If on the same computer, one can also run the clientSNFS application and continue to make directories, create and remove files:

```
./clientSNFS -mount NetDirectory/ -port 6590 -address 127.0.0.1
cd NetDirectory/
echo "Sample file one" > file1.txt
mkdir newdirectory
cd newdirectory
echo "another file" > file.txt
rm file.txt
cd ..
```

This use case demonstrates several of possibilities with our file system. You can create, open, and read directories and files. Several of base unix functions are also available for the user to use with our file system, such as “echo”, “cat”, “vi”, and many other applications. Users are even able to remove files as we’ve implemented `unlink`, just not directories.

Essentially, our SNFS supports most basic file system operations, such as the following:

```
create,open,write,close,truncate,opendir,readdir,releasedir,mkdir,getattr,unlink
```

IV. Group Member Responsibilities

We shared many of the responsibilities over the project, meeting over several of days to plan and code our implementation. Nick Palumbo was the lead programmer, providing the direction and skill in implementing most of the core functionality. Eric Cuiffo, Val Red, and Aiser Sheikh shared roles in helping develop some of the functions, debugging the code, and preparing the final implementation. In addition, Eric implemented error-checking. Val Red and AIser Sheikh also authored the report, to which Nick and Eric were contributors.

V. Conclusion

In our implementation, we learned a lot about communication over network protocols, file systems, and memory management; we applied all of this in creating a basic file system that worked over network between a server and client similar to Samba (SMB) utilizing simple-but-effective `char*` implementations and manipulations with a FUSE-assisted client program that users can utilize to remotely view and operate files on the server.

Overall, we successfully implemented our ***Simple Network File System (SNFS)*** with core functionalities that enable basic file systems to be executed on a remotely mounted drive between a server and a client application over a network.